1980

Principles of Program Design Induced From Experience With Small, Public Programs

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Report Number:
80-337

Principles of Program Design Induced from Experience with Small, Public Programs

Douglas Comer

CSD-TR-337

Computer Science Department
Purdue University
West Lafayette, Indiana 47907

April, 1980

April 15, 1980

Abstract

The art of programming is taught, learned, and practiced as if programs are disposable, personal objects owned solely by the programmer. This paper uses examples to illustrate why real software is neither personal nor disposable; it shows how even simple programs are shared by others. From the examples, the paper extracts four principles for program development and demonstrates how they lead to increased productivity. Finally, it draws conclusions about programming practices and the education of programmers.

anything worth doing is worth doing well

-- anon

we do not know how to teach "good" programming habits

-- P. Wegner

1. Introduction:

Why is programmer[1] productivity low? Why are reliable programs difficult to create or modify? Why do programmers ignore the work of others and build each program from scratch? These questions have prompted recent research into the specification, design, implementation, testing, and maintenance of production software (eg., [Der76, Bel79, Ivi77, Par79]). Such research efforts, which are collectively referred to as "software engineering", aim to improve programmer productivity and increase the reliability, correctness, and cost effectiveness of the final product. Researchers have studied guidelines, techniques, and tools to aid in the development process. The varied approaches range from rigorous mathematical analysis and proof [Dij76, Mil73] to management procedures [Dal77, Bak72].

Despite research efforts to make software manufacturing a simple engineering process, it remains a complex art. Many projects still fall short of design goals, while others are delivered incredibly late. Many programmers cannot produce reliable products, and very few build on the work of others. More astonishingly, professionals who exhibit talent for producing useful, innovative systems seldom seem to know how they learned to do what others cannot. Usually, such successful individuals relate a series of battles they had with computer systems, and simply say that they learned a little from each.[2]

Battling computer systems extracts a heavy toll with little payoff. Programmers waste time and energy to discover a few simple facts. Many of the battles could be avoided altogether if the programmer adopted the correct attitude about programming and followed a few basic principles. The key to avoiding battles is sharing. Sharing each other's experiences and work increases programmer productivity and reduces frustration.

---

[1] The term "programmer" will mean anyone who designs, implements, or maintains computer programs.

[2] In fact, [Yoh74] states that "people are taught how to code; programming is learned only by bitter experience."

Before programmers can share each other's work, they must have confidence in it. Unfortunately, most programs are not designed for sharing; they contain hidden restrictions, dependencies, and flaws. This paper proposes a strengthened attitude toward programming that goes beyond mere stylistic conventions. It develops four principles of programming, and asserts that following them is necessary for producing software that can be shared.

2. Private vs. Public Software:

Brooks [Bro75] claims that there are three types of programs: plain vanilla programs, program products, and systems products. Plain vanilla programs are what students write -- they need not be reliable, portable, documented, or maintained. Most importantly, programmers think of writing vanilla programs as a private communication between the programmer and the machine; the program is thought of as a personal object owned solely by the programmer. Program products, while not necessarily more complicated than vanilla programs, are public. They have to be correct, reliable, documented, and maintained because users who know nothing about the code depend on the output. Systems products are larger and more complicated than program products. Because of the sheer size and complexity, they cannot be designed, implemented or maintained by a single individual. The production of a systems product is among the most challenging of human endeavors.

The process by which programmers are trained instills the attitude that programs are personal property by focusing exclusively on tools and techniques for building small, vanilla programs rather than on guidelines and procedures for constructing and maintaining public software [Ral80]. Beginners learn to write short, virtually useless programs that only serve to illustrate particular programming language constructs. Because of the limited time available, advanced training fails to correct the misconception: one learns to write private programs which are discarded as soon as they run. The point here is that most young programmers emerge from their formal education with poor programming habits and attitudes ingrained. They assume that software is personal property, something that is not shared. When faced with a large systems product, they must adapt to both the fact that it will be public as well as the fact that it is large and complex.

A programmer who first faces the issues of maintaining public (albeit small) software will be better prepared to appreciate and use the tools necessary to tackle large, complex software. For example, at our departmental computing facility, each new programmer is given a small, public program as a first assignment. The programmer has a chance to learn what it means to maintain public software without

jumping directly into operating system maintenance.

The next section presents two small, public programs which illustrate the differences between private and public software and show what a programmer can learn from working with a small program. The remaining sections list four principles of program development that have been extracted from experiences with small programs, and reviews them in light of the examples.

3. Experience with Small Programs:

Using two small, public programs as examples, this section illustrates how software can be shared. The examples are: Lister, a program to paginate and format a file, and Grader, a program to maintain classroom grades.

Lister:

Lister began as one of several utility programs written for use in a batch environment where program source and data files, created using an editor, were stored on disk in a nonstandard format. Because the files had a nonstandard format, they were not accessible by running programs -- they could only be concatenated together and submitted as a batch job. For example, to obtain a hardcopy listing of a file, one had to write a program that copied its input to its output, concatenate the program and the file together, and submit the result as a batch job.

The first version of Lister was designed to do little more than copy its input to its output. It consisted of about 10 lines of PL/I code to duplicate its input, inserting a page number and the date at the top of each page. One could invoke Lister, name one or more files to be listed, and submit the result with a single command, making Lister almost as convenient to use as other system commands. In spite of the convenience, the program itself was trivial.

After some time, it became apparent that Lister lacked desirable features. Because most printers recognized only upper case, the lower case characters in a file were usually lost. In addition, files often contained identification and sequence information in columns 73-80, making listings of them difficult to read. The second version of Lister compensated for these problems by translating all characters to upper case and listing only the first 72 characters of a line. It recognized lines of the form %key=value as commands, however, to allow users to turn off translation or change the line length.

In addition to the program itself, the author maintained a document describing the use of Lister, including some examples. As users asked about the formatted listings,

they were referred to the documentation. In a few months, the use of the Lister file began to increase. Shortly, there was a user population that began to depend on Lister.

Users began to suggest additions, changes, and improvements. Successive versions of Lister had more commands, and performed more formatting functions. By version 10, Lister allowed the user to change the page numbering; add headings; right justify text; direct output to a line printer, card punch, or another file; center text; underscore; change the page length; and include source files by name. The program had been rewritten from scratch at least 3 times in PL/I, SNOBOL4, and Assembler language (the latter was necessary to handle included files). The documentation had grown from 1 page to over 14 pages.

The uses of Lister had changed, too. Instead of one user, there were many. Instead of one or two runs during the week, Lister was invoked every day, usually close to 20 times. And instead of listing existing files, users began to create files that only Lister could recognize and format. A dozen or more student term papers were prepared using Lister as the sole formatting program.

The change in use forced a change in the way Lister was maintained. Since others depended on Lister remaining relatively stable, new versions had to be thoroughly tested before they were installed. To test each new version, it was made available as Listerx. Those users who were anxious to try new features invoked Listerx instead of Lister, and were helpful in testing as well as providing feedback on the design. After a period of testing, Listerx moved to Lister and the new version became the "production" version.

Grader:

Like Lister, the Grader program began as a small, private program for use in computing classroom grades. The first version, written in SNOBOL4, supported a handful of commands to allow the user to enter student's names, identifiers, and grades; to compute weighted averages; and to print the results. Documentation for the program was contained in comments in the source file. When another professor asked about Grader, he was referred to the program source with the warning that he had better check the input carefully because the program did little to validate the data it received.

Despite the simplicity (and lack of adequate input validation), the popularity of Grader soared. Part of the popularity can be attributed to the lack of any reasonable competition: although many students and faculty claimed to have their own grading programs, none was documented or maintained. Part of the popularity, however, was due to

commands which made Grader convenient and flexible. For example, it accepted the grades -1 for "incomplete" and -2 for "omitted", and readjusted the specified weights on an individual basis to ignore the incomplete and omitted work when calculating a weighted average. At the end of the semester, instructors could change all incomplete grades to zero and recompute the weighted averages easily.

It became apparent that Grader needed to be rewritten and expanded, so version 2 was designed and implemented in Pascal. While the second version did support more commands, the major differences arose because it became a program product. Grader 2 checked the input carefully to find and report mistakes and nonsense. All input, including numbers, were read as characters; numbers were converted to internal numeric form after they had been examined for errors. Grader also required the user to declare a maximum value for each homework, quiz, or examination score, and verified that individual scores fell in the correct range as they were entered.

As with Lister, users have suggested extensions and improvements to grader over the past three years. Version 7 has 38 commands which allow one to change headings, footings, rearrange columns of output, omit highest or lowest grades in a group, add and drop students, print the grades, sort the listing, display scores as a bar chart, and even to dump the grade matrix in a format convenient for input to other programs. The source file has grown from 300 lines to 2800 lines, and the user population has grown from one user to several dozen.

As a user population grew, changes to grader had to be considered more seriously. The first version was used to assign grades to 35 students, and every calculation was verified with a calculator. Now, well over a thousand student grades are assigned each semester based on calculations performed by Grader (including complex adjustment of weights for individual students mentioned above). The procedures for modifying the current version (repairing problems) and for installing a new version (making user visible changes) are more complex. Files prepared as input to one version of Grader must be acceptable to later versions or professors could not depend on the program. Of course, new versions are announced and made available for some time before they replace the production version, and the files are locked to prevent simultaneous access and update.

4. Principles of Program Development:

Lister and Grader come to mind as examples of the kind of program product from which programmers learn the differences between public and private software. While both programs were easily managed by one programmer, they both have

the essential ingredients that distinguish them from simple vanilla programs. From such experiences, one can extract the following principles:

The Principle of Use: Programs Will Be Used By Others.

This principle sounds so simple that almost everyone agrees with it at first. Despite its simplicity, programmer training instills an attitude against this principle. Most programmers assume that they will write plain programs unless they are told otherwise. Yet any program worth writing is useful in some way. Plain programs should be thought of as the exception, not the rule. Programmers should plan to make software reliable from the outset instead of beginning with a plain program and trying to add robustness as problems surface.

The experiences with Lister and Grader described above demonstrate how users appreciate and use even simple programs that are documented, correct, and reliable. It might be argued that Lister and Grader are atypical because they represent programs which provide general services of interest to many users. Almost any program, however specialized, will be useful to someone besides the programmer who created it. For example, the author wrote a set of programs to enumerate a restricted class of trie index as part of his research. The programs were so specialized that it seemed obvious that no one, including the author, would ever use them again. In a surprising coincidence, a graduate student from another department needed a program to build trie indexes a few years later, and was able to lift code directly out of the original programs. Naturally, the program details had long been forgotten, so without comments in the programs to document the syntax of the input as well as the details of the algorithm, sharing would have been impossible.

Of course, not all software will become public -- the term fluffware has been applied to one class of programs that are not used by anyone except their creators. Fluffware comprises those (usually quite trivial) programs that one pieces together rapidly, uses once, and then discards. For example, to find a pattern in a text file, one might devise a 5-line SNOBOL4 program and run it interactively without bothering to save the source program. Outside of fluffware, there is little that programmers keep entirely for themselves. Even small utility programs like Lister find their way around and eventually become public. One is forced to conclude that even though it may be simple or specialized, software that is worth keeping should be thought of as public (the public may consist of the programmer looking at the program long after it was written).

To follow the principle of use, one should

- Plan from the start to make programs public. This will save hours of debugging unreliable, incorrect programs when others start using them.

- Document a program or throw it away. This will save hours of explaining to others how to use programs (or reading code to find out yourself).

- Design software to be convenient for the uninitiated. This will save hours of interpreting the documentation.

- Label all output; echo all input. This will save hours when users come to you for help.

If programmers design, implement and maintain all software as if it is production software, their documentation and programming habits improve. They begin to choose better names, comment code, and write more reliable programs. In the beginning, programming with others in mind takes time (Brooks [Bro75] estimates that a program product requires 3 times the effort of a plain program). After a while, designing programs to be shared becomes habitual. One recognizes common pitfalls and problems and forms a set of standard solutions. Because programmers can depend on, and share each other's work, less time is wasted reinventing the wheel. In the long run, less time spent in repairing, explaining, and improving old programs leaves more time to devote to new ones. Unfortunately, only a few programmers take this principle seriously enough to benefit from making small programs correct, reliable, and documented.

The Principle of Misuse: Programs Will Be Abused.

Users, sometimes the programmers themselves, supply the most unlikely input to programs. Empty files, binary files, very large files, object programs, letters in place of digits, digits in place of letters, excessively long lines, and other syntax errors are common. In addition, syntactically valid input will sometimes cause overflow, underflow, division by zero, table overflow, or subscript out of range problems. Finally, users can make subtle errors that cause unexpected output. For example, in Lister it was possible to specify that both a heading and page number should be printed at the same location on the page.

Another form of abuse occurs when users attempt to use a program for something other than what is was designed to do. For example, even though Grader provided only integer valued grades, one professor multiplied every grade by 1000 before entering it in order to obtain values accurate to 3 decimal places. Everything worked fine until he asked for a bar chart to be printed. The program exceeded the output

line limit trying to print a listing with over 100000 lines (one line for each grade from 0 to 100000).

If one takes the point of view of a user, proper techniques for handling the errors become clear. The program should respond with a reasonable message for any possible input, including an empty input file. Grader follows this precept by listing all input and augmenting error messages with a pointer to the exact trouble spot. Naturally, error messages should be phrased so that a user who has never seen the source code can understand the problem; messages should never refer to variable names.

Error correction is more difficult and less important than error detection. If the program does attempt error correction, it should always call attention to amended or inserted input, or possibly incorrect (inaccurate) output that results. For example, an early version of Lister mistakenly truncated lines on the right edge of a page without telling the user, and an early version of Grader accepted fractional input but rounded to the nearest integer without telling the user. Both actions, which were intended to make the input more flexible, led to incorrect output. In both cases the program made the worst mistake possible by presenting output that looked reasonable, contained no warnings, but was incorrect. Later versions corrected these problems by informing users whenever input was altered.

To summarize, the programmer should:

- Give a reasonable output for any input.
- Keep the program from terminating abnormally (eg., from printing a "dump").
- Call attention to corrected (altered) input or inaccurate output.

The Principle of Evolution: Programs Will Change.

Inevitably, one will improve, extend or modify all programs. This principle applies to the most trivial looking programs as well as complex ones. On one hand, errors may crop up when a program does not live up to the advertised specifications. On the other hand, the use and needs of even an error-free program change gradually over time. Features may be added to extend a program's capability, unused features may be deleted to make the implementation more efficient, or existing features may be modified. In anticipation of change, one should expend energy to make the code readable and modular.

Belady [Bel79] calls the phenomenon of evolution "the law of continuing change" and explains some of the causes.

Parnas [Par77] describes specific ways programmers can plan modules for ease of expansion and contraction, and Kernighan and Plauger [Ker78] give detailed rules for programming style and documentation. Suffice it to say that modules should be designed to ease modification, and that the source code should be commented and uniformly styled.

Looking at this issue from a user's point of view, one can see that public software should not change without warning. Rather, user visible changes should be collected together into numbered versions. The version numbers should appear in the source code, documentation, in the object deck, and on the output to provide a link between the running program and the source from which it came. Similarly, minor repairs and changes that do not affect users should be collected together into revisions. The convention of numbering programs as version v.r, where v is the version number and r the revision, works nicely. Beginning with version 0.0, those versions with numbers less than 1.0 can be used while the program is written, making version 1.0 the first released version.

Users need to be warned of imminent version changes, and should have a reasonable assurance that the new version is tested. Users also need access to the documentation for new versions before the update can be made. Finally, the procedure for actually changing the files must be considered carefully: the system may not provide adequate protection against interference between the programmer who updates a file and users who want to read or execute it.

In summary, programmers who plan for change will:

- Make programs readable.
- Write modules to make extension and contraction easy.
- Collect user-visible changes into numbered versions.
- Collect repairs in numbered revisions.
- Keep the documentation current.
- Exercise caution when updating public files.

The Principle of Migration: Programs Will Move To New Machines.

Since the need for most software outlives the machine on which the software runs, one should expect that programs will eventually run in a different environment than the one in which they are created. Interesting and useful programs usually migrate to new machines rapidly, sometimes without the owner's knowledge.

As hardware becomes less expensive, portability will become even more important. Unfortunately, many programmers still think of their task as that of instructing a machine. They take advantage of the nuances and quirks of a particular machine to gain efficiency or reduce the programming effort. To insure portability, programmers must learn to avoid machine details instead of exploiting them; they must think of programs as problem solutions instead of instructions to a machine.

Once programmers accept the principle that programs will be transported to new environments, they work to:

- Write programs to solve problems, not to instruct machines.

- Use standard programming language features.

- Isolate and comment all machine dependencies.

Lister and Grader contrast sharply in portability. Grader, written in Pascal, has moved to several machines and compilers without major effort. Lister, on the other hand, was coded in assembly language, so the result of the hours of work that went into writing and maintaining it were left behind when the author moved to a new computing environment. Of course, some of the ideas have been incorporated into a new formatting program (written in Pascal this time), and others are already provided as utilities in the new environment. The fact remains, however, that the original code for Lister would still be maintained and used if it had survived the transportation process.

5. Summary and Conclusions:

Programmers grow through three phases of programming. First, they learn to write small, private programs. Second, they work with small program products in order to master the techniques of designing, implementing, and maintaining public software. Finally, they are ready to grapple with large, complex system products. This paper has argued that emphasis has been placed on the first phase incorrectly, and that it should be shifted to the second. Moreover, the paper demonstrated the benefits to be gained by making reliable, correct, documented, production programs the rule rather than the exception.

The paper has argued that the design, implementation, and maintenance of public programs can be learned through experience with small programs. Moreover, forcing programmers to jump from small, vanilla programs to large, public systems software in one step confuses the problems of public software with the problems of large software.

In order to design programs for sharing, the programmer must remain conscious of the following principles:

1. Programs will be used by others.

2. Programs will be abused.

3. Programs will change.

4. Programs will move to new machines.

and the specific guidelines for programmers that follow from them. The experiences with Lister, Grader, and other production software, showed how successful programs have been built from these principles, and how programs can fail when they are not followed.

Universities, responsible for training many programmers, cannot continue enforcing the notion that software production consists of writing small, useless programs. Young programmers must learn to deal with public, production software as a way of life. It should be clear that notions and attitudes necessary for good software engineering cannot be relegated to a single course; they must pervade the curriculum, or students will walk away with the impression that public software is the exception rather than the rule. During the past few years, the software industry as well as universities have adopted the attitude that style and structure are important parts of program production. We must now act on the premise that plain vanilla programs are as unacceptable as poorly styled ones.

Acknowledgement: The author would like to thank Walter Tichy and Peter Denning for several suggestions.

References

[Bak72] F. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal 11:1 (January 1972), 56-73.

[Bel79] L. Belady and M. Lehman, "The Characteristics of Large Systems," in Research Directions in Software Technology, P. Wegner, ed., MIT Press, 1979, 106-138.

[Bro75] F. Brooks, The Mythical Man-Month, Addison Wesley, 1975.

[Dal77] E. Daly. "Management of Software Development," IEEE Trans. Soft. Eng. SE-3:3 (May 1977), 230-242.

[DeR76] F. DeRemer, and H. Kron, "Programming-in-the-Large vs. Programming-in-the-Small," IEEE Trans. Soft. Eng., SE-2:2 (June 1976), 80:86.

[Dij76] E. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

[Hor77] J. Horning and D. Wortman, "Software Hut: A Computer Program Engineering Project in the Form of a Game," IEEE Trans. Soft. Eng., SE-3:4 (July 1977), 325-330.

[Ivi77] E. Ivie, "The Programmer's Workbench -- A Machine for Software Development," Comm. ACM 20:10 (Oct. 1977), 746-753.

[Ker78] B. Kernighan and P. Plauger, The Elements of Programming Style, McGraw-Hill Book Company, 1978.

[Mil73] H. Mills, "How to write Correct Programs and Know it," IBM Corp. report FSC 73-5008, 1973, Gaithersburg, Md.

[Par72] D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," Comm. ACM 15:12 (Dec. 1972), 1053-1058.

[Par79] D. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Trans. Soft. Eng. SE-5:2 (March 1979), 128-138.

[Ral80] A. Ralston and M. Shaw, "Curriculum '78 -- is Computer Science Really that Unmathematical?" CACM 23:2 (February 1980), pp 67-70.

[Yoh74] J. Yohe, "An Overview of Programming Practices," Computing Surveys 6:4 (December 1974), pp 221-243.